# Reproducibility through containers

*Robert G. Clapp*

## ABSTRACT

In order to achieve truly reproducible results the underlying software architecture must be captured. Docker containers, a lightweight alternative to a virtual machine, can be used to snapshot all software dependencies and allow anyone to reproduce an author's results with minimal effort. We demonstrate the effectiveness of Docker containers in several contexts including reproducible research, computer labs, and writing LaTeX documents.

## INTRODUCTION

Reproducible research is one of the fundamental building blocks of scientific advancement. Claerbout (1990) and Schwab et al. (1996) designed a framework for authors to follow to make their work reproducible. Their basic concept was to introduce generic targets of `build` (build all results), `view` (view the results), `burn` (remove the results), `clean` (remove all intermediate files) that each author would define for each paper. A limitation of their approach is that when their underlying software dependencies changed, reproducibility (or even the ability to compile) is not guaranteed. Fomel and Claerbout (2009) went a step further incorporating papers into the software building process. As a result, in theory, papers stay reproducible because any bugs introduced into the software are caught and fixed in the build process. This approach, in addition to producing an ever larger, and more complex building/debugging requirement, still makes assumptions that all possible underlying software requirements have been tested/debugged.

The difference in development and production environments is a well known problem in the broader software development field. Virtualization, which creates a virtual machine running within a system, is one approach. Virtualization has drawbacks such as slow provisioning, performance degradation, and a large memory footprint when running multiple virtual machines on a single server. A newer approach is containers (Wikipedia, 2016). The current leading container approach are 'Docker containers'. In the context of reproducible research, a Docker container will include: the rules to build a paper, the software needed to build the paper, the underlying operating system which the user built the paper on, and how that underlying operating system was constructed.

In this paper we demonstrate how to use Docker containers to build a reproducible environment. We demonstrate their utility by using them for report articles, labs,

and even building documents.

## INTRODUCTION TO DOCKER CONTAINERS

At a very basic level a container can be thought of as a virtual machine. You have an additional operating system that is taking a part of a host's resources. In general, the operating system of the virtual machine exists in the computer's memory instead of on disk. The advantage of the virtual machine concept is that we can construct the perfect environment for my application to run in. In terms of reproducible research, once we've tested that the code works in a given virtual machine, that working state is preserved forever. We can distribute the virtual machine image to anyone and it will simply work without much, or any, effort on their part. In addition, we can run multiple virtual machine images on a single host, each running completely independently.

The complete independence of each virtual machine is also one of its major drawbacks. Imagine using virtual machines to do a parallel task, because each virtual machine is completely independent each will have its own complete copy of the operating system. Containers work a little differently. We normally think of a filesystem as consisting of a series of directories and files. The directories and files may sit on different disks or servers but only a single version of a given directory exists for a given machine. As we make a Docker image, we are building up the filesytem in a series of layers. Each command in the Docker build process takes a difference between the state of the filesytem before and after a build command. Each layer of the Docker image is read-only. By default any changes we make to the Docker filesytem while running a given Docker image is making changes to a temporary additional filesystem layer. When we exit, all of these changes will be lost (we will discuss read/write filesystem layers later). When running multiple Docker virtual machines the read-only layers will be shared between the Docker instances resulting in a much smaller memory footprint.

In addition Dockers run on the host operating system, allowing it to share a lot of the host resources. As a result while a virtual machine can take minutes to start a Docker often starts in less than a second.

### Building a Docker image

A Docker image is built from a `Dockerfile`. A `Dockerfile` consists of a series of commands to build the Docker image. As an example we are going to step through a `Dockerfile` that builds SEPlib on a Centos-7 machine.

```
From centos:7
MAINTAINER Bob Clapp <bob@sep.stanford.edu>
```

The beginning of my `Dockerfile` indicates that we want to start with another Docker image, in this case, CentOS version 7 (the colon is how you indicate a version number to Docker). SEPlib uses SU's library for reading SEG-Y headers so before we build SEPlib we are going to build SU. The CentOS image is quite small, with the minimum number of packages. In order to build SU we need to install sum X11 libraries, make, gcc, and wget using yum. The `RUN` keyword indicates system commands that need to be run to build the image.

```
RUN yum -y install make libX11-devel libXt-devel gcc wget; yum -y clean all
```

After we've installed these packages we need to create a directory for SU and then download and compile it. We need to do a little bit of work because SU wants to have this section be interactive.

```
RUN mkdir /opt/SU && cd /opt/SU ; \
   wget ftp://ftp.cwp.mines.edu/pub/cwpcodes/cwp_su_all_44R1.tgz  &&\
  cd /opt/SU ; tar xf cwp_su_all_44R1.tgz;cd /opt/SU/src   &&\
  cd /opt/SU/src ;touch cwp_su_version LICENSE_44_ACCEPTED MAILHOME_44 &&\
  cd /opt/SU/src;  echo "echo boo"  >chkroot.sh &&\
  cd /opt/SU/src; chmod 777 chkroot.sh   &&\
  cd /opt/SU/src; CWPROOT=/opt/SU make install xtinstall &&\
   rm -rf /opt/SU/cwp_su_all_44R1.tgz /opt/SU/bin
```

Docker compares the filesystem before and after each command. By combining all of the installation steps and cleanup on a single line we reduce the overall size of my Docker image. Once we have the parts of the SU that we need to install some additional software packages the SEPlib uses that SU does not.

```
RUN yum -y install make automake autoconf libtool csh git \
 libXt-devel libX11-devel libXaw-devel  gcc gcc-gfortran flex &&
  yum -y clean all
```

We need to download, compile, and install SEPlib.

```
RUN mkdir /opt/SEP &&\
    git clone http://zapad.Stanford.EDU/bob/SEPlib.git /opt/SEP/src && \
    cd /opt/SEP/src; autoreconf -vif  &&\
    cd /opt/SEP/src; ./configure --prefix=/opt/SEP  --with-su=/opt/SU && \
    cd /opt/SEP/src;  make install &&\
    cd /opt/SEP/src;  make clean
```

Finally, we are going to add the environmental variables SEPlib needs to the root user's environment.

```
RUN echo export PATH=$PATH:/opt/SEP/bin >> ~/.bash_profile &&\
    echo export SEP=/opt/SEP >> ~/.bash_profile &&\
    echo export SEPINC=/opt/SEP/include >> ~/.bash_profile &&\
    echo export PYTHONPATH=/opt/SEP >> ~/.bash_profile
```

Once we've written my `Dockerfile` we can build my Docker image using

```
 docker build -t rgc007/seplib:8-centos .
```

where `-t rgc007/seplib:8-centos` indicates that the tag `-t` for this image is `seplib` version `8-centos` for the Docker account `rgc007`. The `.` indicates the directory where the `Dockerfile` and any additional files we might add to my image exist. After the image is built we can run:

```
docker run -it rgc007/seplib:8-centos /bin/bash
```

The run command will check to see if the image exists locally. If not, it will it will attempt to download the layers needed for the Docker image from from the Docker account `rgc007`. Any layers that don't exist locally will then be downloaded. Finally it will give me a bash shell within the container containing a full version of SEPlib. we can push this image to Docker by using the command `docker push rgc007/seplib:8-centos`. Once Docker is installed the reader can reproduce the SEPlib build by typing `make buildSEPRepo` or enter the Docker image by typing `make enterSEP`.

## USING DOCKERS

There are many uses for containers. Below we discuss four examples of using Dockers in a research university environment. The Dockerfile described above sets up an image with a basic SEPlib environment. My reproducible research Docker is going to begin by inheriting my SEPlib image.

```
From rgc007/seplib:8-centos
MAINTAINER Bob Clapp <bob@sep.stanford.edu>
```

We could follow the same procedure described above to build and enter my Docker but we wouldn't be able display any X11 graphics. There are several different options to get graphics working. The one we are going to choose is to add a local ssh server to my Docker. Using a ssh server approach has the advantage of working on Linux, MacOSX, and with the right ports and an X11 client, Windows. First we need to install ssh, passwd, and xauth.

```
RUN yum -y install  openssh-server passwd  xauth; yum clean all
```

Next we are going to add to the Docker a script that creates a non-root user, sets its password, and copy roots environment. In addition it is going to set the SEPlib datapath to
`tmp`.

```
ADD start.sh
```

where start.sh contains

```
#!/bin/bash

__create_user() {
# Create a user to SSH into as.
useradd user
SSH_USERPASS=newpass
echo -e "$SSH_USERPASS\n$SSH_USERPASS" | (passwd --stdin user)
echo ssh user password: $SSH_USERPASS
echo datapath=/tmp/ >~user/.datapath
cp ~root/.bash_profile ~user/.bash_profile
}

# Call all functions
__create_user
```

To run the ssh daemon we need to create an additional directory and generate an ssh key.

```
RUN mkdir /var/run/sshd
RUN ssh-keygen -t rsa -f /etc/ssh/ssh_host_rsa_key -N ''
```

We are then going to run the `start.sh` script.

```
RUN chmod 755 /start.sh
RUN ./start.sh
```

We are then going to add to my new Docker a local folder `derivative` containing a Makefile and a simple code that applies a first derivate to a 2-D field.

```
ADD derivative /home/user
RUN chown -R user /home/user
```

Finally we need to indicate that to enter this image that to enter it one should use ssh.

```
ENTRYPOINT ["/usr/sbin/sshd", "-D"]
```

Once we've built this image (e.g. `docker build -t rgc007/testReport`), We can start the image using

```
docker run -d -p 22  -t rgc007/testReport
```

in this case we are running the Docker in detached mode and mapping port 22 of the Docker to an available port on my local machine. We can get what port my Docker image mapped port 22 to using `docker ps`. Finally, we can login to my Docker image using `ssh -Y user@localhost -p XXXX` where XXXX is the port number I found from the `docker ps` command[1] We use the password `newpass` to login and We are now in a complete linux environment where we can use make `build, burn, clean, view`. The reader can build the reproducibility Docker by typing `make buildSEPRepo`. You will need to enter the password `newpass` at the prompt.

Another use, in an academic research context, is to use a Docker to build LaTex environment. The Docker image consists of a series of read-only file system layers. As a result even though we can modify a file within the Docker filesystem those changes will be lost once the Docker image is stopped. For the reproducible research example above the read-only nature of the file system is sufficient. When writing a paper we need to have some persistency in our filesystem. The solution to what Docker refers to as Docker data volume. The basic concept is to map a directory from our host system into a directory inside Docker images. Any changes made in the directory of the Docker image is seen on the host and vice versa. To create a Docker volume we simply add an option to the `docker run` command line option of the form `--volume=/local/path:/docker/path` where `local/path` refers to a directory on the host machine and `/docker/path` refers to the directory in Docker image. The reader can build the SEP LaTeX environment by typing `make buildSEPTeX` and enter the image by typing `make enterSEPTeX`.

SEP has a series of four classes that it requires every student to take. One of the challenges are making sure that peculiarities of a given SEPlib release, compiler, user environment, etc. do not distract from the purpose of the lab. Dockers solve this problem. Once a Docker image is built and works once it will work without needed additional changes. This also allows us to provide our labs to the world without the challenges of completely setting up their environment. The lab Docker combines elements of the LaTeX and reproducible research Docker. The Docker can be built using `make buildSEPlLab` and entered using `make enterSEPLab`, again using the `newpass` password.

Another interesting use for a Docker is to distribute single software executables. The entrypoint of a Docker can be running a unix command. When used in conjunction with a Docker data volume, theoretically a full processing system could be

[1]It is useful to add -Y -o UserKnownHostsFile=/dev/null -o StrictHostKeyChecking=no to avoid having to edit your known host file.

written using Docker containers. A `CMD` is added the `Dockerfile` specifying what program should be run and any potential arguments. As an example typing `make showModel` will first build a 3-D synthetic model and then display it using SEPlib's 3-D viewer.

## CONCLUSIONS

Dockers represent a more complete step along the path of reproducible research. They can be used to capture not only the user's code but the user's full environment. Dockers also provide effective solutions for code distribution, computational labs, and building LaTeX documents.

## REFERENCES

Claerbout, J. F., 1990, Active documents and reproducible results: SEP-Report, **67**, 139–144.

Fomel, S. and J. F. Claerbout, 2009, Reproducible research: Computing in Science & Engineering, **11**, 0005–7.

Schwab, M., M. Karrenbach, and J. Claerbout, 1996, Making scientific computations reproducible: SEP-Report, **92**, 327–342.

Wikipedia, 2016, Operating-system-level virtualization — Wikipedia, the free encyclopedia. ([Online; accessed 23-September-2016]).